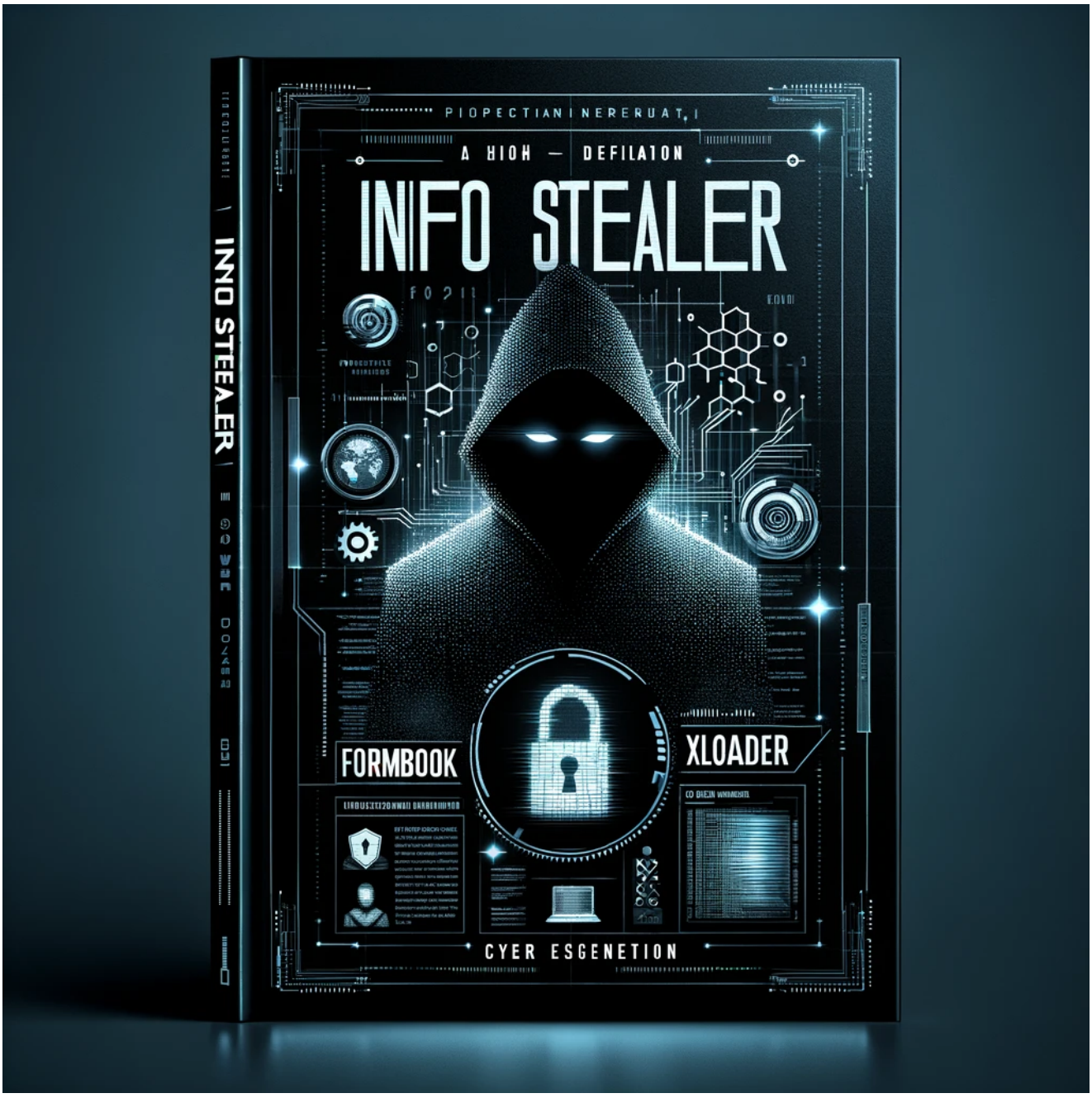
Layers of Deception: Analyzing the Complex Stages of XLoader 4.3 Malware Evolution

medium.com/@shaddy43/layers-of-deception-analyzing-the-complex-stages-of-xloader-4-3-malware-evolution-2dcb550b98d9

Shayan Ahmed Khan April 25, 2024



Shayan Ahmed Khan

--

XLoader, an advanced evolution of the **FormBook** malware, stands out as a highly sophisticated cyber threat renowned for its dual functionality as an **information stealer** and a versatile downloader for malicious payloads. Noteworthy for its resilient nature, xLoader constantly adapts to the latest and most intricate **evasion techniques**, making it a formidable challenge for cybersecurity defenses. Its notoriety is heightened by its role as a commercial **Malware-as-a-service** solution, enabling cybercriminals to tailor and deploy the malware for diverse malicious activities. The malware's continuous evolution and ability to elude detection emphasize the critical need for robust cybersecurity measures to counter its intricate and multifaceted attacks, which target both individuals and organizations alike.

Key Findings:

- Xloader uses a similar initial dropper as some of the other infostealers like Remcos RAT and Agent Tesla. The initial dropper is a dotnet executable file, which contains multiple embedded which are extracted and decrypted at run-time to launch the payload which is the actual malware. The payload is launched using in either itself or another running process, depending upon the configuration of the initial dropper.
- Xloader is written in native low level asm/c language. There are nostrings, imports and libraries found in this payload. Native assembly with the combination of c language already makes it than other infostealers like Remcos, Agent Tesla, NanoCore etc.
- It uses advance techniques that detects if the malware is running in an analysis environment. The usage of advanced techniques makes sure that, are not easily bypassed as simply as patching a jump condition or return condition.
- It uses a encryption/decryption algorithm with additional subtraction operations.
- Xloader uses hashing algorithm for its strings, libraries and APIs to hide its internal working.
- Xloader's core malicious functions are all encrypted that are decrypted at-run time and assembly is renewed or regenerated after all anti-vm checks have been bypassed and a key has been generated.
- It uses a clean copy of manually mapped into its memory which bypass all hooks for ntdll APIs. It uses Native APIs for its malicious activities which are hidden from EDR solutions. The process is known as "" according to FireEye.
- Xloader adds persistence using Run Registry Keys and copying itself in Program Files (x86).
- It escalates privileges only for copying itself in the Program Files (x86) and adding persistence. The privilege escalation is achieved by abusing DllHost.exe and COM objects.
- Xloader relies heavily on process injection. It infects multiple processes in its execution and even migrate to a different process.
- It uses a combination of decoy C2 servers and made significant effort to hide its real C2.

- Xloader is not just an infostealer. It also works as a form grabber. Inline hooks are injected into multiple victim processes to grab information before encryption is performed.

Check out my Github Repo for Malware Analysis Series!!!

Overview

XLoader emerges as an exceptionally sophisticated infostealer and form grabber malware, distinguished by its adept use of advanced defense evasion techniques to maintain stealth and resilience. Beyond its evasive maneuvers, XLoader incorporates a myriad of anti-VM techniques, strategically avoiding execution in analysis environments. This malware's primary objective is data exfiltration, achieved through the theft and capture of sensitive information from a broad spectrum of applications, including browsers, email clients, FTP clients, and instant messaging apps. Notably, XLoader is designed to operate seamlessly across a variety of platforms, amplifying its threat level. Its multifaceted attack flow encompasses a strategic and systematic approach, making it a potent tool for cybercriminals seeking to compromise both individual users and organizational systems. The constant evolution of XLoader underscores the need for robust cybersecurity measures to counter its intricate and adaptable nature.

Xloader Attack Chain

Threat Report: XLoader 4.3

This section of the report provides a detailed technical analysis of **Xloader 4.3** malware. The flow of this report will be in order of steps that I performed during my analysis. This is one of the most complex pieces of malware that I have analyzed, and there are so many stages to its execution. I have tried to cover as much as possible in the given time, but if some things remain unanswered then I apologize beforehand. Now let us dive down into the technical details and internal workings of Xloader 4.3 previously known as **Formbook** infostealer.

Initial Detonation:

Starting with the initial detonation of xloader. I have detonated the malware in my isolated analysis environment in the presence of procmon, wireshark and other such analysis tools. **Nothing happened!!!** Which likely suggests that there are anti-analysis techniques in the malware. I tried detonating the malware again but this time, I had **renamed** my analysis tools and the execution started.

· Process tree shows that it started another instance of itself.

· Multiple DNS & HTTP request are sent to different domains.

· Deleted itself

· Request are sent through explorer.exe

Few of the resolved domains are listed below:

- hxxp:\\www.twin68s.online
- hxxp:\\www.cicreception2023.org
- hxxp:\\www.morubixaba.com
- hxxp:\\www.gestionamostualquiler.org
- hxxp:\\www.superios.info
- hxxp:\\www.bocahkota.xyz
- hxxp:\\www.lolisex77.top
- hxxp:\\www.fhsbfjbsljsdfsd.xyz
- hxxp:\\www.mifurgoentuangar.fun
- hxxp:\\www.necessarymusthave.shop
- hxxp:\\www.abk-importexport.com
- hxxp:\\www.adoniadou.com
- hxxp:\\www.delret.tech
- hxxp:\\www.humidlandscaping.com
- hxxp:\\www.wlkwinn.net
- hxxp:\\www.8ai.ooo
- hxxp:\\www.minevisn.com
- hxxp:\\www.moheganmart.com
- hxxp:\\www.jacksonmoddy.com

Stage 1: Dropper

The initial dropper is a dotnet executable. It is similar to what other infostealers or RAT uses for dropping their payloads like Agent Tesla or Remcos RAT. The first step is always static analysis, which extracts suspicious strings for me and provide insight to the malware.

The extracted strings suggest 3 main points:

- Dropper is obfuscated that loads other assemblies at run-time
- Further resources are inverted to avoid signature-based detection
- Must have more than 1 assemblies

In the initial dropper, there is a lot of junk code added to divert the focus of analyst. The few lines of malicious code are spread through the whole code.

The relevant lines of code shows that malware is loading binary from 3 different resources:

· Quartz which is also reversed

· Versa

· Zinc

These 3 are the malicious resources that are combined and loaded at run-time for further execution. After going through a lot of junk code, I came across the line of code that resolves this assembly at run-time and create instance of resource followed by loading the first method using **System.Activator** class.

Since, stage1 malware resolves assemblies at run-time and activate the method from resolved assemblies therefore static analysis is not possible ahead of this step, so I shifted to dynamic analysis.

- The runtime binary that has been loaded can be seen in the modules window.
- The name of runtime generated binary is . In the code, the malware is invoking the first member returned by the GetExportedTypes which means the first member of exports would be executed.
- We can locate the first function in the pendulum binary and set the breakpoint ahead to stop and debug it.

There are further binaries being resolved from the resource of first loaded DLL which is Pendulum. In the modules tab, we can trace which dlls are being added and keep following through.

- Another binary that is being loaded at run-time from the resource of pendulum is the which could be seen in the modules window. This binary undergoes gzip decompression and loaded using Activator class.
- This binary contains a few methods called “ and ” which performs some kind of decryption of another third resource which will also be loaded on runtime.

- The last resource that has been decrypted and loaded is called .
- In the method of ParsingState, it could be seen that a method from this assembly is being called for further execution of malware.

- We can also see the names of classes and methods that are being called from this assembly in the locals. Using this information, we can then setup another breakpoint in the method and continue debugging the 3rd resource.
- Again, we can explore the third binary and setup a breakpoint on the function that it tries to call.

We have now entered the method called by the previous dll. This binary is highly obfuscated with random variable and class names. Normally, what I do is that I check if a deobfuscator like de4dot or some other tool is able to deobfuscate such a binary. If it is possible then I patch the resource and continue my debugging with the deobfuscated version. But in this

case, it is very tricky because this resource is dependent upon two other binaries that are being called first and to patch all these will be such a headache. So, I decided to move forward with the obfuscated version and see if I could understand what it is doing from the local variables and return values.

- I kept stepping over and checking the variables and function returns.
- It skipped most of the flags but then I stepped over a function and a return value shows that another binary has been returned. The MZ bytes (4D 5A) could be seen in the array.

- It confirms that this malware might perform some kind of injection or dump the binary in a file and execute it as a 2nd stage malware.
- I stepped into a function that is obfuscated but it looks like it is performing , as the malware opens itself in a suspended state and ready to inject in the address space of this process.

- Stepped over few of the functions while checking RWX memory region of the process
- At one point it reserved the memory and then started writing shellcode into that memory in chunks
- It changes the execution of base image to the injected shellcode and finally resume the process using ResumeThread API.

- This is the exact behavior of process hollowing.
- I dumped this shellcode to analyze the malware separately as a second stage payload.
- The stage2 malware is the real xloader payload.

Stage2: Xloader 4.3

Xloader is an infostealer malware that is the updated version of Formbook malware. It is sold on dark web for cheap prices with a MaaS architecture (Malware-as-a-Service). The authors of this malware put great effort in adding latest defense evasion techniques.

- Xloader aka Formbook is written in pure native assembly with a combination of c language
- The entropy is very high which suggests that there is embedded code or it might be packed
- There are 0 libraries, imports, strings found in this payload
- There are no valid strings other than the DOS message

- The start of malware is fairly simple, it loads some necessary libraries before going to the malicious code
- It also performs some other kind of computations, probably decompressing some of its malicious code
- After the calculations, I came across a call to edx which leads to an unidentified code

- The instruction moves the program flow to a set of native assembly which is unidentified by IDA at this moment
- This means that, the code to which edx register now points was not understood by IDA which indicates that it might be encrypted at first
- From there the execution of real formbook payload starts

- IDA resolves this chunk of assembly at run-time to continue debugging this dump.
- This is one of the many anti-analysis techniques added in the xloader payload.

After going through the newly resolved chunk of code, my program exited without doing anything else. I understood that there are anti-analysis techniques involved in this malware. So, my battle started with defeating anti-analysis techniques provided in the section below.

Defeating Anti-Analysis:

TAKE # 1: FAILED

- In first take, I simply changed the jump condition to divert the program from exiting the malware to continue with the actual program flow
- Changed the zero flag from 1 to 0 which sets the condition appropriately to let the program continue

- It continues the program, however it throughs exception right after stepping over a few functions.
- This patch will not work
- The malware is dependent upon the values that this flag is setting somewhere

TAKE # 2: FAILED

The configuration object:

- Xloader payload initializes a configuration object on which it bases most of its execution flow
- The configuration obj is initialized with FFFFFFFF value and after that each function contributes to it.
- Some encrypted values are pushed onto this configuration object.

- The first function, saves lots of encrypted strings or hash codes. The purpose of these will be cleared later on in the execution
- Next to FF values, the base address of executing malware is saved
- On the third line another address is stored which is actually the address of function from ntdll. This will be used to laod further libraries

- I stepped over each function and monitored changes in memory side by side.
- Every function is contributing to the conf obj.
- The function in the screenshot below is loading a clean ntdll in the memory and saves it address on the conf obj
- Also, it is setting value in anti-vm flags that starts from the 45th element of the conf obj.
- The address of injected ntdll in memory starts on and similarly in the 4 bytes after 24th element we have the address of injected ntdll saved.
- The flag value of 1 is also set in anti-vm flags.

- Continuing with the execution.
- It checks other anti-vm checks
- Like taking snapshot of running processes and filtering out if any of those processes are listed by the malware
- In the screenshot, we can see that it detected in running processes

  After performing some of the anti-vm checks, it updated the flags on anti-analysis bytes as shown in screenshot below:

- The last function is matching the anti-vm flags with the sequence it requires to progress.
- As can be seen in the screenshot, my sequence doesn't match to what it should be,
- It means the malware has either or tools like or some other parameter
- Therefore, the program exits.

- So, in take # 2 of defeating anti-reverse engineering or anti-vm techniques, I simply patched the sequence of these flags in the memory to the required sequence.
- Patching memory, and moving onto the execution should work, because these flags are being used somewhere ahead in the program. So, simply changing the conditional jump would always crash the program.
- However, in case of memory patch, these values would be continued in the program and this issue should be fixed.

- Patched the memory and now it goes back to the condition which is true
- However, something is wrong here.
- Because the names of the dll being searched is very weird.
- Now I understand, that these sequences of bytes are being used in a decryption algorithm to decrypt the names of libraries and APIs.
- But since I patched the bytes in memory, it should have been able to decrypt accurately which it is not. That means that the sequence is used somewhere else before performing the anti-analysis check.

- I let the malware continue and again it crashed, because it was not able to decrypt its configuration and hence looking for encrypted dll names.

- So that means, I might be missing some important function and because it is detecting the debugger, it would be skipping some important function.

TAKE # 3: PASSED

- In third take, I have debugged a lot of the code and finally, found the function over which the program was skipping because of a single flag condition not being met.
- So, I changed the values of condition to allow it to execute as well as changed the value of register that was being pushed to the .
- In my environment, there were always 3 flags that were changed. The value on the third element was 0 however it should be 1, and the two elements at 11,12th position.
- I also know that those two were changed because of procmon and other such analysis tools. So, it is easier to just change the name of procmon and continue.
- Instead of applying memory patches, I have changed the values at run-time before they were pushed onto the memory stack and , the malware executed perfectly without any exceptions.

- Now this time, I stepped over the function that loads libraries and instead of encrypted names, the full names of libraries have been seen and successfully loaded as can be seen in procmon.
- I let the program continue without any other interaction and the debugger exited with status code 0, which means now there is no exception.
- However, it still hasn't performed all the functionality which indicates there are ahead.

I found a very good resource, that explains all the flags that previous formbook version looked for in its analysis. Luckily in the latest xloader, it is still using a similar approach and we can map those flags easily. The following slide shows all the anti-analysis flags that the xloader uses in its configuration.

Reference: https://www.botconf.eu/botconf-presentation-or-article/in-depth-formbook-malware-analysis/

Decryption/Deobfuscation Routine:

Xloader relies heavily on encryption and obfuscation to avoid being detected from EDR solutions. There is multi-layered encryption performed on its code. The APIs are all hashes, the string and libraries are also hashes. Even the hashes are encrypted in the conf obj. The core functions of xloader are all encrypted and decrypted at run-time after anti-analysis checks are cleared.

Decrypting Library Names:

- The decryption routine starts, I stepped through the next function after anti-vm checks have been cleared and it looks like the anti-vm flag bytes are used as decryption seed value.
- The library names are being decrypted one by one.

  These libraries are then loaded by the native function

Decrypting API Names:

- Some of the APIs that are being decrypted suggests that it looks for further
- 1. LookupPrivilegeValueW
- 2. SeDebugPrivilege
- 3. AdjustPrivilegeToken

Computing String Hashes:

- There is a hashing algorithm used for strings, apis etc.
- It loads all the string hashes and compare the running processes with each hash value, if it finds any such process, it adds desired value on the anti-vm flag on conf obj.
- In the screenshot below, it is checking the process name hash with the value of pre-defined set of hashes that it stored.

- The hash value that is it is comparing to is which in hex is (0xCDC7E023).
- I have checked 32-bit hashing algorithms by calculating the hash of procmon and found the hashing algorithm that it uses.
- It uses hashing for its strings

All the hashes that it checks are listed below:

Computing API Hashes:

- Similar to strings hashes
- The APIs that are being loaded from injected () are also called by hashes instead of names
- This method makes detection very hard even for manually analyzing the malware.
- The malware loads all exports of ntdll one by one and computes the CRC-32/BZIP2 hash of those apis then compares it with its decrypted hashes.
- If a match is found, then it retrieves the address and call the function,

- I wrote a little script that does the same, I provide the hash and it searches in a list of commonly used strings,apis,paths etc, computes their hashes and then compares with the provided hash to check weather a match has been found or not.
- Here in this case, the hash matched on API call, so malware will exit the loop and continues to retrieve the address and then call the api.

- It manually searches for the address of desired API and calls it, this way the debugger is also not able to detect which API is being called.
- In the screenshot below, I have opened another instance of same dll in IDA with symbols and we can see the hex value that is being pushed onto eax register is the same.

- I know the hashing function, so instead of stepping through this native assembly of hundreds of functions in a loop, I have just setup the breakpoint on that function by writing IDA python script and just continuing again and again to see the decrypted APIs
- The List of APIs that I found are listed below:

Decrypting Core Malicious Functions:

- The malware decrypts its core functions at run-time and then jumps to those functions continuing the execution flow.
- Xloader sets up a function by and and other starting instructions but below these all bytes are encrypted.
- In previous versions of formbook, the core malicious functions could be identified by the magic bytes of 48909090, 49909090 etc.
- However, in the latest xloader 4.3 these starting bytes are random.
- After the anti-vm checks and establishing the RC4 decryption key. These functions are decrypted at run-time and the execution flow jumped to the decrypted assembly.
- IDA resolves the decrypted bytes and recreates assembly instructions to continue.

Understanding the detailed technical methodology of decrypting these encryption and obfuscation techniques. This following blog by **zscaler** is an excellent resource.

Technical Analysis of Xloader's Code Obfuscation in Version 4.3

Analysis of the new variant of Xloader information stealer malware that identifies itself as version 4.3, released on…

www.zscaler.com

Partially Decrypted Shellcode:

- Stepped over a few functions and it looks like it reads itself and most likely trying to inject itself in some other process
- The malware is now preparing for another binary to inject further. As can be seen in the screenshot of the dump that I found in the memory
- This memory dump is memory region in itself as can be seen in the process hacker

- I stepped over a few functions while monitoring the memory region.

- The malware is decrypting the shellcode from the binary
- Only plain shellcode is left without MZ headers
- This is the 3rd stage xloader which is partially decrypted
- I dumped the binary from memory and run a FLOSS string search on it which provides some useful insights

Process Enumeration:

XLoader uses to get information of all running processes in the system and then enumerates one-by-one checking and matching hashes with its own hash values stored in conf obj.

Process Injection:

Xloader Injection Overview:

Xloader stage2 performs two process injections:

- in a random running process to start the win32 victim process in suspended state
- migrate itself into win32 suspended process and resume

Injection # 1

- Another memory has been reserved in the malware with RWX memory region.
- I have dumped this new region and extracted the strings
- It has a single static string which contains the name of the target process

- It means that this shellcode is used for starting the process which is randomized on every execution.
- Xloader selects these binaries from SysWOW64 directory, which are 32-bit processes
- It injects this shellcode in one of the above enumerated running processes, which in my case is a 64-bit IDA that I had opened along with my debugger.

  This is also one of the anti-analysis techniques used by xloader. It doesn't directly open the process itself but injects shellcode in some random process which in turn opens the SysWOW64 randomized binary in a suspended state and then retrieves its process information and continue with the execution.

- In Ida64, the shellcode is injected which starts the process and return the process information back to stage2 malware of xloader.
- The RWX memory region could be seen in IDA64.
- This is just a after opening the target process in suspended state.

Injection # 2:

- The second injection is performed in the chkdsk.exe (randomized SysWOW64 binary)
- There are two buffers injected in the chkdsk.exe.
- 1 buffer of 180KB and other of 40KB
- Since this malware is performing so many injections, it is very difficult to keep track of everything so we got an idea of creating a tool for detecting process injections.
- I would like to give special thanks to , for creating this tool in short period of time which is very useful in detecting injections of such malware.

Tool link:

- The smaller buffer contains the original chkdsk.exe bytes.
- I also found the function that writes shellcode in the empty buffer.
- This is also a shared memory region between the formbook payload and victim process of chkdsk.exe
- Because the buffer is simultaneously being written in both processes.

- Here in xloader payload, the memory region is also being written simultaneously
- This is the same partially decrypted shellcode that I have displayed above, with most of the decrypted strings.
- From here onwards, the stage3 of formbook will be executed.

- Finally, after resuming the suspended process in chkdsk.exe
- It exits using ExitProcess API

Stage 3: Partially Decrypted Xloader 4.3

Before resuming the thread on injected process. I have attached x32dbg to the victim process to continue debugging further. In the EAX register, the address of xloader injected code is already set by stage2 malware. So, I just jumped to address in disassembly and added breakpoint on it. Then from the stage2 malware I allowed the malware to continue hence resuming the thread on stage3. Stage2 malware has exited and we have debugger attached to the entry point of stage3 malware which I will continue from here. This whole execution flow is very similar to stage2 malware. So, I will move forward with only key details in this section:

Defeating Anti-Analysis:

- Xloader has decrypted some of its functions and now migrated to the process (which was in previous examples)
- Before resuming the thread, I've attached debugger to the injected process and continued my analysis from there.
- This is the same cycle being repeated first.

- I have to defeat anti-analysis techniques again
- Similar to stage2 I have bypassed anti-analysis techniques again and correct sequence of bytes have been generated as highlighted below

Decryption/Deobfuscation:

- This injected stage3 payload performs the same initial steps.
- It performs anti-vm techniques and checks
- Decrypt further library names and load using LdrLoadDll
- Decrypt API names and match hashes. Finally load those APIs from the injected fresh copy of
- A few of the APIs that it uses for Process Injection are resolved:
- 1. LookupPrivilegeValue
- 2. SeDebugPrivilege
- 3. NtAdjustPrivilegeToken

Indicator Removal:

- It will delete the stage2 malware with following sequence of APIs
- 1. NtCreateFile
- 2. NtQueryInformationFile
- 3. NtReadFile
- 4. NtClose
- 5. ZwDeleteFile

Process Injection:

- The next series of APIs being used are:
- 1. NtCreateSection
- 2. NtMapViewOfSection
- 3. NtAllocateVirtualMemory
- 4. NtOpenProcessToken
- 5. NtQueryInformationToken
- 6. ConvertSidToStringW
- 7. NtAllocateVirtualMemory
- It is preparing another shellcode to inject further in some process. There are a few more RWX sections created in the memory of infected process

System Information Discovery:

- It retrieves the system information from the Registry like the of OS etc
- 1. NtCreateKey
- 2. NtQueryValueKey

Dynamic Library/API resolution:

Loading libraries using

Process Enumeration & Injection:

- Looks like the next injection will be in .
- It enumerates all the process by looping through the list of processes returned by
- 2. NtCreateMutant
- 3. NtCreateSection
- 4. NtMapViewOfSection
- 5. NtDelayExecution
- 6. NtAllocateVirtualMemory

Bot registration:

- The data it collects and sends in the first request is provided below:
- The Magic word: XLNG
- Bot ID: 202293EF
- Xloader Version: 4.3
- OS: Windows 10 Enterprise x64
- Username: base64_encoded

Stealer:

- Xloader is an infostealer and form grabber.
- After registering the device, it looks for all the things it could steal from the victim
- There are a large number of email clients, browsers, ftp clients, messaging apps that it tries to look for in different paths to fetch and steal the data

- If it finds anything, it then tries to steal that data
- Like in case of chrome, it founds login data and it will fetch the data using sqlite3 queries
- It uses winsqlite3.dll to extract passwords
- The query is
- It decrypts that data using . from the key found in local state

- If it finds anything, it creates a file in temp folder with the static name of
- If the file exists already, it first deletes the previous one and then write new with the updated date.
- Reads the file by the following API sequence
- 1. NtCreateFile
- 2. NtQueryInformationFile

- 3. NtReadFile
- 4. NtWriteFile

Targeted Processes & Applications

Decrypted Functions:

- A lot of data is hidden at first because of encrypted functions
- Similar to stage2 malware, the stage3 version also have encrypted functions in it
- Those are decrypted at run-time
- Those functions also contain encrypted hex-based strings for targeted processes
- The strings for targeted applications and paths are pushed onto stack at run-time.

Privilege Escalation:

- Privileges are escalated by abusing the dllhost.exe and COM objects
- It keeps trying to copy the stage2 malware in Program Files
- If proper privileges are not provided, it then uses explorer to write stage2 malware in temp and by abusing dllhost, it copies the malware to Program Files

Persistence:

- After the malware is copied in Program Files
- It achieves persistence by adding Run Registry Keys
- It uses the API

Setting Inline Hooks:

- Xloader also works as a form grabber
- It sets inline hooks in targeted processes for stealing plaintext data from the parameters of the functions
- The data stolen from victim processes is saved in a shared memory between 3 processes
- 1. Victim Process
- 2. Stage3 Malware
- 3. Explorer
- The xloader is stuck in a loop here
- On every loop, it does the following:
- 1. Enumerates all running processes
- 2. Set inline hooks in targeted processes if found (by injecting code)
- 3. Steal clipboard data
- 4. Tries to create a file in program files
- 5. Adds registry in RunKeys

- 6. Send a POST & GET request on one of the resolved c2 servers through It has an injected payload in explorer.exe that it uses for exfiltrating stolen data.

Setting hooks

Ref:

References: